Exploration of the Dijkstra’s Algorithm

Through Network Routing

Katherine Wilsdon

Idaho State University

Universal Classes or Methods

Calculate Distance

The calculate_distance( ) method determines the Euclidean distance between two nodes via the distance formula, Figure 2. The distance is multiplied by 100 to normalize the distance from a decimal to a whole number.

Node

The node class was created to hold information such as the node_id, distance, previous, location, and neighbors, shown in Figure 3. The node_id, location, and neighbors are directly pulled from the network.nodes data structure to ease look up. The distance and previous node are updated according to Dijkstra's algorithm.

Distance

The distance class in Figure 4 is used to store data returned by the Unsorted Array / Binary Heap implementations as well as the getBestShortestPath class. The list returned by Dijkstra's algorithm is stored into shortest_paths. The list returned by getBestShortestPath is stored in the best_shortest_path list.

```
def calculate_distance(p1, p2):
    dist = math.sqrt((p1.x() - p2.x())^2 + (p1.y() - p2.y())^2)
    return dist*100
```

Figure 1: Calculate Distance Pseudocode.

```
# Calculates the Euclidean distance between two nodes
def calculate_distance(self, point1, point2):
    dist = math.sqrt((point1.x() - point2.x()) ** 2 + (point1.y() - point2.y()) ** 2)
    return dist * 100
```

Figure 2: Calculate Distance Algorithm.

```
class Node:

    def __init__(self, node_id, distance, previous, loc, neighbors):
        self.node_id = node_id
        self.distance = distance
        self.previous = previous
        self.loc = loc
        self.neighbors = neighbors
```

Figure 3: Node Class Implementation.

```
class Distance:
    def __init__(self):
        self.shortest_paths = []
        self.best_shortest_path = []
```

Figure 4: Distance Class Implementation.

Dijkstra's Algorithm

Unsorted Array and Binary Heap

Both implementations of Dijkstra's algorithm make the queue by inserting the source node into the array. A list of the shortest paths is created containing the nodes returned by delete_min. While the queue size is not empty, the node with the smallest distance is deleted from the queue and appended the node to the list. For all the edges on the deleted node, the Euclidean distance is calculated between the deleted node and its neighboring node. The edge is skipped if the neighboring node has been previously deleted. If the neighboring node has not been visited before (dist = ∞), the neighboring node is inserted into the queue and previous is set to the deleted node. If the calculated distance between the deleted node and its neighboring node is less than the neighboring node's current distance, the distance of the neighboring node is decreased and previous is set to the deleted node. See the Space and Time Complexity section for runtime analysis.

```
def dijkstra():
    make_queue()
    create list of shortest paths nodes
    while currentSize > 0:
        min = delete_min()
        list.append(min)
        for all edges of min:
            dist = calculate_distance (u, v) + dist(u)
            if previously deleted:
                continue
            else if node has not been visited:
                insert(v, dist)
                v.previous = u
            else if dist < v.dist:
                decrease_key(index of v, dist)
                v.previous = u
    return list
```

Figure 5: Dijkstra Pseudocode Algorithm.

```
# Finds the shortest paths between nodes in the graph
def Dijkstra(self):
    self.make_array()
    shortest_paths = []
    while self.currentSize > 0:
        min_node = self.delete_min()
        shortest_paths.append(min_node)
        # For all edges of min_node
        for i in range(len(min_node.neighbors)):
            neighbor_id = min_node.neighbors[i].dest.node_id
            # If the neighbor was already deleted
            if self.unsortedArray[neighbor_id].distance == -1:
                continue
            dist = self.calculate_distance(min_node.loc, self.network.nodes[neighbor_id].loc) + min_node.distance
            # If the neighbor is not visited, insert into the unsorted array
            if self.unsortedArray[neighbor_id].distance == float("inf"):
                self.insert(neighbor_id, dist)
                self.unsortedArray[neighbor_id].previous = min_node
            # Update the distance value if a shorter distance
            elif dist < self.unsortedArray[neighbor_id].distance:
                self.decrease_key(neighbor_id, dist)
                self.unsortedArray[neighbor_id].previous = min_node
        self.unsortedArray[min_node.node_id].distance = -1
    return shortest_paths
```

Figure 6: Unsorted Array Implementation of Dijkstra's Algorithm.

```
# Finds the shortest paths between nodes in the graph
def Dijkstra(self):
    self.make_heap()
    shortest_paths = []
    while self.currentSize > 0:
        min_node = self.delete_min()
        shortest_paths.append(min_node)
        # For all edges of min_node
        for i in range(len(min_node.neighbors)):
            neighbor_id = min_node.neighbors[i].dest.node_id
            dist = self.calculate_distance(min_node.loc, self.network.nodes[neighbor_id].loc) + min_node.distance
            # If the neighbor was already deleted
            if self.pointerArray[neighbor_id] == -1:
                continue
            # If the neighbor is not visited, insert into the binary heap
            elif self.pointerArray[neighbor_id] == float("inf"):
                self.insert(self.network.nodes[neighbor_id], dist)
                self.binaryHeapArray[int(self.pointerArray[neighbor_id])].previous = min_node
            # Update the distance value if a shorter distance
            elif dist < self.binaryHeapArray[int(self.pointerArray[neighbor_id])].distance:
                self.decrease_key(neighbor_id, dist)
                self.binaryHeapArray[int(self.pointerArray[neighbor_id])].previous = min_node
    return shortest_paths
```

Figure 7: Binary Heap Implementation of Dijkstra's Algorithm.

Unsorted Array Algorithms

Make Array and Insert

The make_array( ) function inserts the source node into the array shown in Figure 10. The insert( ) method changes the distance at a particular index in the array from infinity to the provided distance, Figure 11. The current size is incremented by one. The time and space complexity is O(1) because indexing into the array and changing the value occurs in constant time.

Decrease Key

The decrease_key( ) method changes the current distance at a particular index in the array to a smaller distance shown in Figure 12. The time and space complexity is O(1) because indexing into the array and changing the value occurs in constant time.

Delete Min

The delete_min( ) deletes the node with the minimum distance, Figure 13. By looping through every index the unsorted array, it first finds the starting index that is not unvisited or previously deleted and proceeds to find the node with the minimum distance. The current size is decremented by one. The time and space complexity is O(|V|) because the algorithm loops through every node in the array.

```
Unsorted Array

def make_array():
    insert(source index, 0.0)

def insert(index, dist):
    dist(unsortedArray[index]) = dist
    currentSize += 1

def decrease_key(index, dist):
    dist(unsortedArray[index]) = dist

def delete_min():
    min_index = 0
    is_found = false
    for i from 0 to len(unsortedArray):
        if is_found and dist(unsortedArray[i]) == -1
            continue
        else if is_found and dist(unsortedArray[i]) < dist(unsortedArray[min_index]):
            min_index = i*2
        else if not is_found and (dist(unsortedArray[i] == inf or dist(unsortedArray[i] == -1):
            min_index += 1
        else if not is_found and dist(unsortedArray[i] != inf or dist(unsortedArray[i] != -1:
            is_found = true
            if currentSize == 1:
                break
    min_node = unsortedArray[min_index]
    currentSize -= 1
    return min_node
```

Figure 8: Pseudocode for the Unsorted Array Class.

```
class UnsortedArray:
    def __init__(self, network, source):
        assert (type(network) == CS4412Graph)
        self.network = network
        self.source = source
        self.unsortedArray = []
        self.currentSize = 0
```

Figure 9: Unsorted Array Initialization.

```python
# Populate the unsorted array with unvisted nodes and insert the source node into the unsorted array
def make_array(self):
    self.unsortedArray = [Node(x, float("inf"), None, self.network.nodes[x].loc, self.network.nodes[x].neighbors) for x in range(len(self.network.nodes))]
    self.insert(self.source, 0.0)
```

Figure 10: Unsorted Array Make Array Algorithm.

```python
# Inserts the distance into the specified index
def insert(self, index, dist):
    self.unsortedArray[index].distance = dist
    self.currentSize += 1
```

Figure 11: Unsorted Array Insert Algorithm.

```python
# Updates the distance of the node
def decrease_key(self, index, dist):
    self.unsortedArray[index].distance = dist
```

Figure 12: Unsorted Array Decrease Key Algorithm.

```python
"""Finds and deletes the min node"""
def delete_min(self):
    min_node_index = 0
    is_found = False
    for i in range(0, len(self.unsortedArray)):
        # If the node was already deleted
        if is_found and self.unsortedArray[i].distance == -1:
            continue
        # If the distance is smaller, set the minimum node index to the current index
        elif is_found and self.unsortedArray[i].distance < self.unsortedArray[min_node_index].distance:
            min_node_index = i
        # Find the starting index for the minimum node index
        elif not is_found and (self.unsortedArray[i].distance == float("inf") or self.unsortedArray[i].distance == -1):
            min_node_index += 1
        elif not is_found and self.unsortedArray[i].distance != -1 and self.unsortedArray[i].distance != float("inf"):
            is_found = True
            if self.currentSize == 1:
                break
    min_node = self.unsortedArray[min_node_index]
    self.currentSize -= 1
    return min_node
```

Figure 13: Unsorted Array Delete Min Algorithm.

Binary Heap Algorithms

Make Heap, Insert, Bubble Up

The make_heap( ) function inserts the source node into the binary heap shown in Figure 16. In Figure 17, the insert( ) function inserts the node at the bottom of the binary heap and relocates this node to the correct position depending on its distance by calling bubble_up( ), Figure 18. The bubble_up( ) function swaps the parent node with the child node when the distance of the child node is smaller. This repeats until the correct position is found. The space and time complexity of bubble_up( ) is log(|V|) because the bottom, child index is halved repeatedly until finding the correct index. make_heap( ) and insert( ) are also log(|V|) because inserting the node into the binary heap is O(1) followed by a call to bubble_up( ).

Delete Min, Siftdown, Min Child

The delete_min( ) method in Figure 19 deletes and returns the node with the minimum distance which is O(1). The min node is replaced with the last node in the binary heap which is also O(1). Then the siftdown( ) function is called to swap the root with its child that has the smaller distance repeatedly until the correct position is found, shown in Figure 20. The time and space complexity of siftdown( ) is log(|V|) because in order to find the child, the parent index is multiplied by 2 repeatedly. The min_child( ) method returns the node with the smallest distance of either the left or right child or zero for no children, Figure 21. The time and space complexity of min_child( ) is O(1) because the algorithm compares the two distances which is constant time. The total complexity of delete_min( ) is log(|V|) because of the siftdown( ) function.

Decrease Key

In Figure 22, the decrease_key( ) function decreases the distance of the provided node and relocates this node to the correct position depending on its distance by calling bubble_up( ),

Figure 18. decrease_key( ) is log(|V|) because decreasing the distance of the node is O(1) followed by a call to bubble_up( ) which is log(|V|).

```
Binary_Heap

def make_heap():
    insert(source node, 0.0)

def insert(node, dist):
    Create node object and insert into binaryHeapArray
    currentSize += 1
    Update pointer array with currentSize
    if currentSize != 1:
        bubble_up(currentSize)

def bubble_up(index):
    i = index
    p = ceil(i/2)
    while i != 1 and dist(binaryHeapArray[i]) < dist(binaryHeapArray[p])
        swap(i, p)
        i = p
        p = ceil(i/2)

def delete_min():
    min = binaryHeapArray[1]
    Update pointer array for min and currentSize node
    binaryHeapArray[1] = binaryHeapArray[currentSize]
    binaryHeapArray.pop(currentSize)
    currentSize -= 1
    if currentSize != 0:
        siftdown(1)
    return min

def siftdown(index):
    i = index
    min_child = min_child(i)
    while min_child != 0 and dist(binaryHeapArray[i]) > dist(binaryHeapArray[min_child]):
        swap(i, min_child)
        i = min_child
        min_child = min_child(i)

def min_child(i):
    if 2 * i == currentSize:
        return i * 2
    else if 2 * i > currentSize:
        return 0
    else:
        if dist(binaryHeapArray[i*2]) < dist(binaryHeapArray[i*2+1]):
            return i * 2
        else:
            return i * 2 + 1

def decrease_key(index, dist):
    binaryHeapArray[pointerArray[index]] = dist
    bubble_up(pointerArray[index])
```

Figure 14: Pseudocode for the Binary Heap Class.

```python
class BinaryHeap:
    def __init__(self, network, source):
        assert (type(network) == CS4412Graph)
        self.network = network
        self.source = source
        # insert dummy node into array for index 0, Binary Heap values start at index 1
        self.binaryHeapArray = [Node(-1, float("inf"), None, QPointF(0.0, 0.0), None)]
        self.pointerArray = [float("inf") for x in range(1, len(self.network.nodes)+1)]
        self.currentSize = 0
```

Figure 15. Binary Heap Initialization.

```python
# Insert the source node into the binary heap
def make_heap(self):
    self.insert(self.network.nodes[self.source], 0.0)
```

Figure 16: Binary Heap Make Heap Algorithm.

```python
# Inserts node at the end of the binary heap and relocates the node to the correct position
def insert(self, node, dist):
    node = Node(node.node_id, dist, float("inf"), node.loc, node.neighbors)
    self.binaryHeapArray.append(node)
    self.currentSize += 1
    self.pointerArray[node.node_id] = self.currentSize
    if self.currentSize != 1:
        self.bubble_up(self.currentSize)
```

Figure 17: Binary Heap Insert Algorithm.

```python
# Relocates the node to the correct position by swapping the parent with the child node
def bubble_up(self, index):
    i = index
    parent = math.ceil(i/2)
    # While i is not the minNode and the distance of the child is less than the distance of the parent
    while i != 1 and self.binaryHeapArray[i].distance < self.binaryHeapArray[parent].distance:
        # Swap child and parent
        self.pointerArray[int(self.binaryHeapArray[i].node_id)], self.pointerArray[int(self.binaryHeapArray[parent].node_id)] = parent, i
        self.binaryHeapArray[i], self.binaryHeapArray[parent] = self.binaryHeapArray[parent], self.binaryHeapArray[i]
        i = parent
        parent = math.ceil(i/2)
```

Figure 18: Binary Heap Bubble Up Algorithm.

```python
"""Deletes the min node and replace it with the last node of the binary heap array,
   and relocates that node to the correct position """
def delete_min(self):
    min_node = self.binaryHeapArray[1]
    # Update the pointer array
    self.pointerArray[min_node.node_id] = -1
    if self.currentSize != 1:
        self.pointerArray[self.binaryHeapArray[self.currentSize].node_id] = 1
    # Puts the last node into index 1
    self.binaryHeapArray[1] = self.binaryHeapArray[self.currentSize]
    # Delete last node
    self.binaryHeapArray.pop(self.currentSize)
    self.currentSize -= 1
    # Relocate the node at index 1 to the correct position
    if self.currentSize != 0:
        self.siftdown(1)
    return min_node
```

Figure 19: Binary Heap Delete Min Algorithm.

```python
# Swaps the root with its smallest child less than the root, repeats until the node is in the correct position
def siftdown(self, index):
    i = index
    min_child = self.min_child(i)
    while min_child != 0 and self.binaryHeapArray[i].distance > self.binaryHeapArray[min_child].distance:
        # Swap parent and child
        self.pointerArray[self.binaryHeapArray[i].node_id], self.pointerArray[self.binaryHeapArray[min_child].node_id] = min_child, i
        self.binaryHeapArray[i], self.binaryHeapArray[min_child] = self.binaryHeapArray[min_child], self.binaryHeapArray[i]
        i = min_child
        min_child = self.min_child(i)
```

Figure 20: Binary Heap Siftdown Algorithm.

```python
# Determines which child has the smallest distance
def min_child(self, index):
    if 2 * index == self.currentSize:
        return index * 2 # Return only, left child
    elif 2 * index > self.currentSize:
        return 0  # No children
    else:
        # Determine the minimum child
        if self.binaryHeapArray[index*2].distance < self.binaryHeapArray[index*2+1].distance:
            return index * 2  # Return left child
        else:
            return index * 2 + 1 # Return right child
```

Figure 21: Binary Heap Min Child Algorithm.

```
# Updates the distance of the node and relocate the node to correct position
def decrease_key(self, index, dist):
    self.binaryHeapArray[int(self.pointerArray[index])].distance = dist
    self.bubble_up(int(self.pointerArray[index]))
```

Figure 22: Binary Heap Decrease Key Algorithm.

Master Methods

Compute Shortest Paths

The computeShortestPaths( ) method determines the shortest paths of the graph from a source node, Figure 23. Depending on whether the use_heap parameter is true or false, the system will create a BinaryHeap object or a UnsortedArray object and run Dijkstra's algorithm on the particular implementation of the priory queue. See the Space and Time Complexity section for runtime analysis.

Get Shortest Path

The getShortestPath( ) algorithm determines the shortest path between the source node and destination node shown in Figure 24. The getBestShortestPath( ) method is called on the BestShortestPath object to return the shortest path. If the destination is not found, unreachable is returned. Next, edges are created and appended to path_edges. The total_length of the path and the path_edges list are returned. The time and space complexity is O(|V| + |E|) because all edges and relevant nodes are visited in order to synthesize the shortest path.

Get Best Shortest Path

The getBestShortestPath( ) parameters are the destination index and a list of the nodes, shortest_paths, returned after running Dijkstra's algorithm, Figure 26. A list is created, and the destination node is found in the shortest_paths. The shortest path is derived by using the previous attribute of the Node to get the preceding node with the smallest distance. The worst case scenario for the time and space complexity is O(|V|) because the algorithm will search through the entire shortest_paths list for the destination node. If the destination node is not found, an empty list is returned.

```
# Determine the shortest paths of the graph using an unsorted array or a binary heap
def computeShortestPaths(self, srcIndex, use_heap=False):
    self.source = srcIndex
    t1 = time.time()
    if use_heap:
        array = BinaryHeap(self.network, self.source)
        self.distance.shortest_paths = array.Dijkstra()
    else:
        array = UnsortedArray(self.network, self.source)
        self.distance.shortest_paths = array.Dijkstra()
    t2 = time.time()
    return (t2 - t1)
```

Figure 23: Compute Shortest Paths Algorithm.

```
# Determine the shortest path between the source node and destination node
def getShortestPath(self, destIndex):
    self.dest = destIndex
    path_edges = []
    total_length = 0

    # Get shortest path
    shortest_path = BestShortestPath()
    self.distance.best_shortest_path = shortest_path.getBestShortestPath(self.dest, self.distance.shortest_paths)

    # If the destination node is not found, return unreachable
    if len(self.distance.best_shortest_path) == 0:
        return {'cost': float('inf'), 'path': path_edges}

    edges_left = len(self.distance.best_shortest_path)-1
    node = self.network.nodes[self.source]
    cur = 1
    while edges_left > 0:
        for next in range(len(node.neighbors)):
            # Determine which neighbor is the next node
            if node.neighbors[next].dest.node_id == self.distance.best_shortest_path[cur].node_id:
                # Add the edge to the list
                edge = node.neighbors[next]
                path_edges.append((edge.src.loc, edge.dest.loc, '{:.0f}'.format(edge.length)))
                total_length += edge.length
                node = edge.dest
                edges_left -= 1
                cur += 1
                break
    return {'cost': total_length, 'path': path_edges}
```

Figure 24: Get Shortest Path Algorithm.

```
BestShortestPath

def getBestShortestPath(dest, shortest_paths):
    Create list
    dest_pos = -1
    for i from 0 to len(shortest_paths):
        insert dest into list
    if dest_pos == -1
        return empty list
    next_pos = dest_pos
    for i from dest_pos -1 to 0, incr -1:
        if shortest_paths[i] node == prev(shortest_paths[next_pos]):
        insert shortest_paths[i] into list
        next_pos = i
    insert source node into list
    return list
```

Figure 25: Best Shortest Path Pseudocode.

```python
class BestShortestPath:
    def __init__(self):
        pass

    # Finds the shortest path between the source node and destination node
    def getBestShortestPath(self, destIndex, shortest_paths):
        best_shortest_path = []
        dest_pos = -1
        # Find destination index
        for x in range(len(shortest_paths)):
            if shortest_paths[x].node_id == destIndex:
                dest_pos = x
                best_shortest_path.insert(0, shortest_paths[x])
                break
        # If destination not found
        if dest_pos == -1:
            return best_shortest_path
        next_pos = dest_pos
        for x in range(dest_pos - 1, 0, -1):
            # Find the previous node and insert the node into the front of the list
            if shortest_paths[x].node_id == shortest_paths[next_pos].previous.node_id:
                best_shortest_path.insert(0, shortest_paths[x])
                next_pos = x
        # Insert the source node
        best_shortest_path.insert(0, shortest_paths[0])
        return best_shortest_path
```

Figure 26: Best Shortest Path Algorithm

Time and Space Complexity

Unsorted Array

As previously stated in the Unsorted Array Algorithms section, the time and space complexity is O(1) for insert( ) and decrease_key( ) and O(|V|) for delete_min( ). In Dijkstra’s algorithm, delete_min( ) occurs |V| times because every node should be deleted unless the path is unreachable. So, the time and space complexity of delete_min( ) is |V| * |V|. For insert( ), each node is inserted into the unsorted array, which |V|, and each node has a number of edges or neighbors, which is 3, that will be checked upon deletion from the unsorted array, |E|. The time and space complexity of insert( ) is (|V| + |E|) * 1. So the overall complexity is $O(|V|^2 + |V| + |E|)$. The delete_min( ) method dominates the runtime, which results in $O(|V|^2)$ complexity.

Binary Array

As previously stated in the Binary Heap Algorithms section, the time and space complexity is O(log|V|) for insert( ), decrease_key( ), and delete_min( ). delete_min( ) occurs |V| times since every node is deleted for a total runtime of delete_min( ) to be |V|log(|V|). With regards to the insert( ) function, each node is inserted into the array for a total of |V| times. Every node has 3 neighbors or edges that are checked upon deletion from the binary heap, |E|. The time and space complexity of insert( ) is (|V| +|E|) * log(|V|). So the overall complexity is O(|V|log(|V|) + (|V| +|E|)log(|V|)). The insert( ) function dominates the runtime, which results in O((|V| +|E|)log(|V|)) complexity.

Verification of Shortest Path

Random seed 42 - Size 20

Because the neighbors of each node are randomly selected, each of the 20 nodes did not have a neighbor to the destination node resulting in an unreachable solution.

Figure 27: Random seed 42 - Size 20, node 7 as the source and node 1 as the destination.

Random seed 123 - Size 200

The unsorted array took 0.006981 seconds to compute the shortest path. But the min heap implementation took 0.001995 seconds which is 3.5 times faster than the unsorted array. The total path cost was 911.081 units.

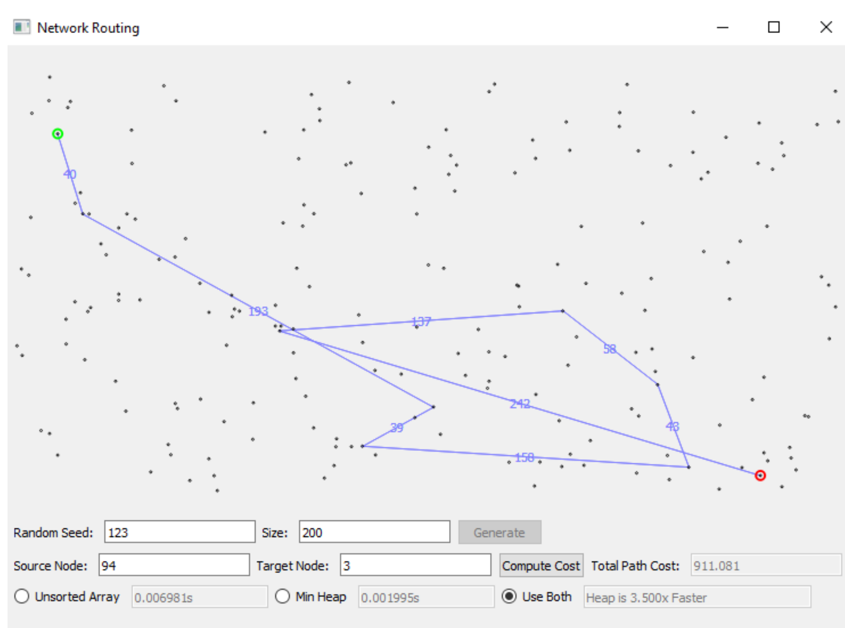

Figure 28: Random seed 123 - Size 200, node 94 as the source and node 3 as the destination.

For Random seed 312 - Size 500

The unsorted array took 0.038896 seconds to compute the shortest path. But the min heap implementation took 0.005984 seconds which is 6.5 times faster than the unsorted array. The total path cost was 1218.803 units.

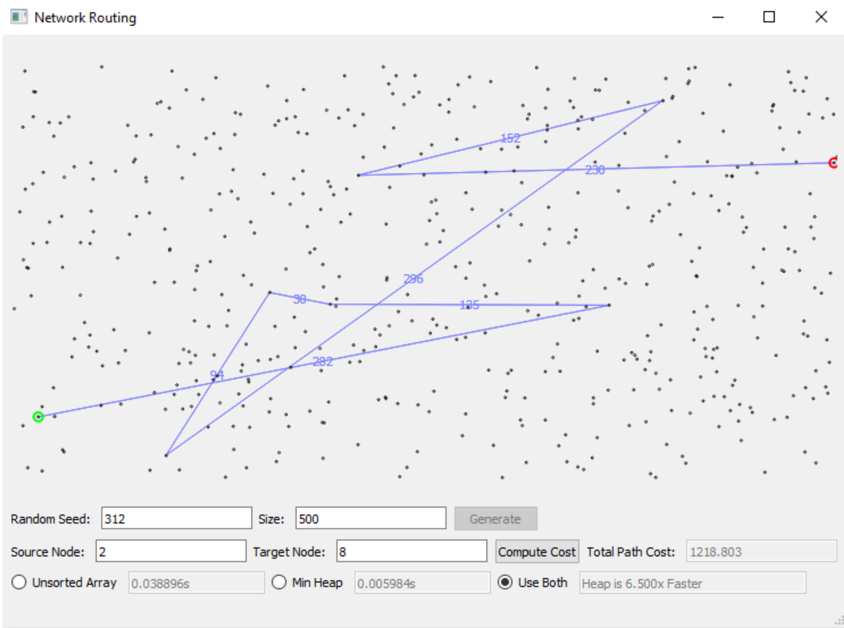

Figure 29: Random seed 312 - Size 500, node 2 as the source and node 8 as the destination.

Results

Network Size of 100

The unsorted array took on average 0.00199 seconds, while the speed of the binary heap was 0.000997 seconds. The binary heap was 2.0 times faster than the unsorted array.

Trial	Size	Random Seed	Source Node	Destination Node	Unsorted Array (sec)	Binary Heap (sec)	Faster
1	100	68	48	29	0.001995	0.000997	2
2		344	97	40	0.001995	0.000997	2
3		575	34	75	0.001993	0.000998	1.998
4		811	53	2	0.001994	0.000998	1.999
5		1134	9	64	0.001994	0.000997	2
				Average	0.001994	0.000997	1.9994

Table 1: Comparison of Unsorted Array and Binary Heap Implementation with a Network Size of 100.

Network Size of 1,000

The unsorted array took on average 0.168 seconds, while the speed of the binary heap was 0.0130 seconds. The binary heap was 12.9 times faster than the unsorted array. The ratio of 1,000 faster / 100 faster or 12.938 / 1.999 is 6.47.

Trial	Size	Random Seed	Source Node	Destination Node	Unsorted Array (sec)	Binary Heap (sec)	Faster	Ratio 1000 / 100
1	1000	21	586	969	0.163562	0.012965	12.615	
2		380	252	168	0.167552	0.012965	12.923	
3		549	376	472	0.16456	0.012965	12.692	
4		797	175	61	0.168537	0.012965	12.999	
5		1000	840	451	0.174533	0.012965	13.461	
				Average	0.167749	0.012965	12.938	6.470941282

Table 2: Comparison of Unsorted Array and Binary Heap Implementation with a Network Size of 1,000.

Network Size of 10,000

The unsorted array took on average 16.7 seconds, while the speed of the binary heap was 0.177 seconds. The binary heap was 94.3 times faster than the unsorted array. The ratio of 10,000 faster / 1,000 faster or 94.354 / 12.938 is 7.29.

Trial	Size	Random Seed	Source Node	Destination Node	Unsorted Array (sec)	Binary Heap (sec)	Faster	Ratio 10000 / 1000
1	10000	38	5977	8089	16.05506	0.170544	94.14	
2		240	5356	206	16.72557	0.172539	96.938	
3		425	6923	9779	17.73148	0.172539	102.768	
4		953	3258	2544	16.94468	0.182512	92.841	
5		1010	3730	1061	15.86856	0.186502	85.085	
				Average	16.66507	0.176927	94.3544	7.292811872

Table 3: Comparison of Unsorted Array and Binary Heap Implementation with a Network Size of 10,000.

Network Size of 100,000

The unsorted array took on average 33 minutes, while the speed of the binary heap was 2.88 seconds. The binary heap was 675 times faster than the unsorted array. The ratio of 100,000 faster / 10,000 faster or 675 / 94 is 7.12.

Trial	Size	Random Seed	Source Node	Destination Node	Unsorted Array (sec)	Binary Heap (sec)	Faster	Ratio 100000 / 10000
1	100000	98	69511	1022	1978.488	3.264095	606.137	
2		192	83887	2306	1919.416	2.272172	705.221	
3		309	94276	32164	1951.571	2.58409	755.226	
4		500	50151	3484	1892.221	2.77757	681.25	
5		975	88265	39652	2213.681	3.525569	627.893	
				Average	1991.075	2.884699	675.145	7.155420415

Table 4: Comparison of Unsorted Array and Binary Heap Implementation with a Network Size of 100,000.

Network Size of 1,000,000

The binary heap took on average 37.4 seconds. The ratios of 100,000 / 10,000 and 10,000 / 1,000 were averaged for a predicted ratio of 1,000,000 / 100,000 of 7.22. The Faster column of the network size of 100,000 was multiplied by the 7.22 ratio to obtain that the binary heap is on average 4,877 times faster. The faster column of the network size of 1,000,000 was used to predict the time for running the unsorted array, which resulted in an average of 50.7 hours or 2.11 days.

Trial	Size	Random Seed	Source Node	Destination Node	Predicted Unsorted Array (sec)	Binary Heap (sec)	Predicted Faster	Predicted Ratio 1000000 / 100000
1	1000000	10	645056	102240	150417.2	34.3512	4378.8	
2		111	616407	152183	196349.3	38.54069	5094.6	
3		330	832314	738718	197779.9	36.25105	5455.84	
4		689	698828	813445	199056.2	40.44682	4921.43	
5		855	308111	51326	169711	37.41446	4535.97	
				Average	182662.7	37.40084	4877.33	7.224116143

Table 5: Comparison of Unsorted Array and Binary Heap Implementation with a Network Size of 1,000,000.

Empirical Analysis

The estimated complexity for the unsorted array implementation is $O(|V|^2)$, which should be $O(n^2)$ or polynomial. From the Figure 30, the data indicates that the Unsorted Array is n^2 because the shape is a parabola. The estimated complexity for the binary heap implementation is $O((|V| +|E|)\log(|V|))$, which is equivalent to $O(n \log n)$. From Figure 31, the data indicates that the Binary Heap is n log n because the shape is close to linear but has a slight curve. So, the empirical data performed as expected.

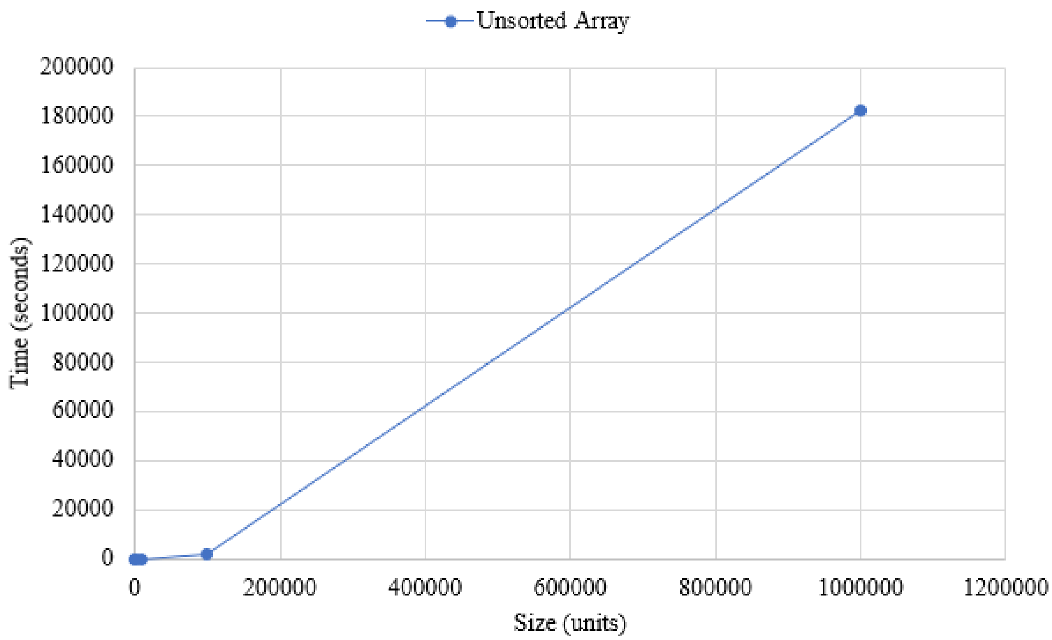

Figure 30: The Empirical Time Complexity for an Unsorted Array.

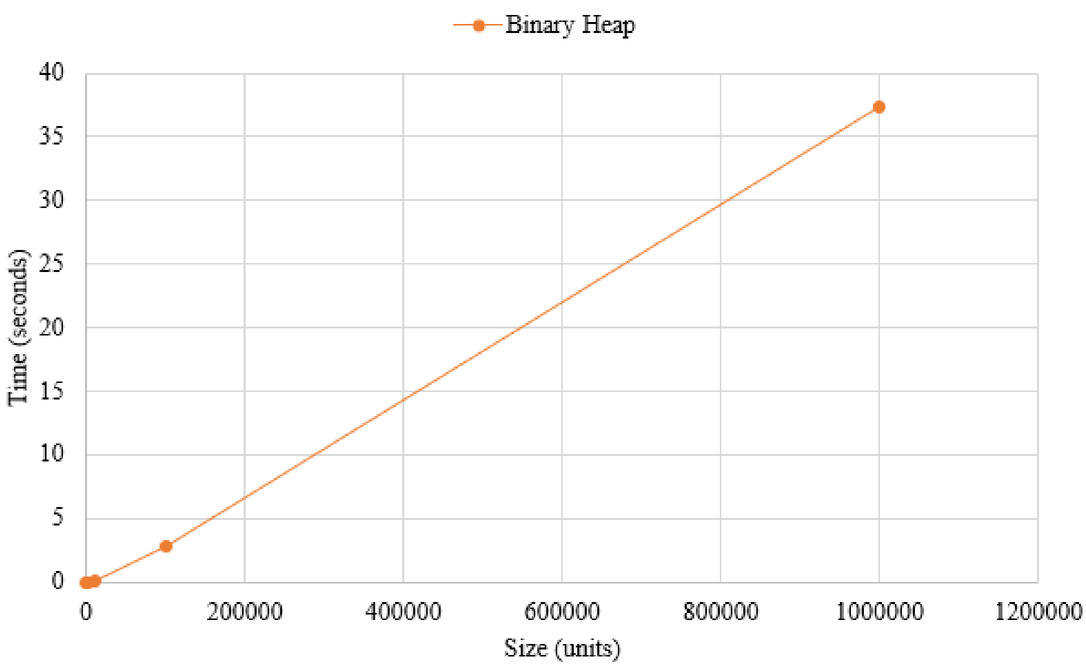

Figure 31: The Empirical Time Complexity for a Binary Heap.

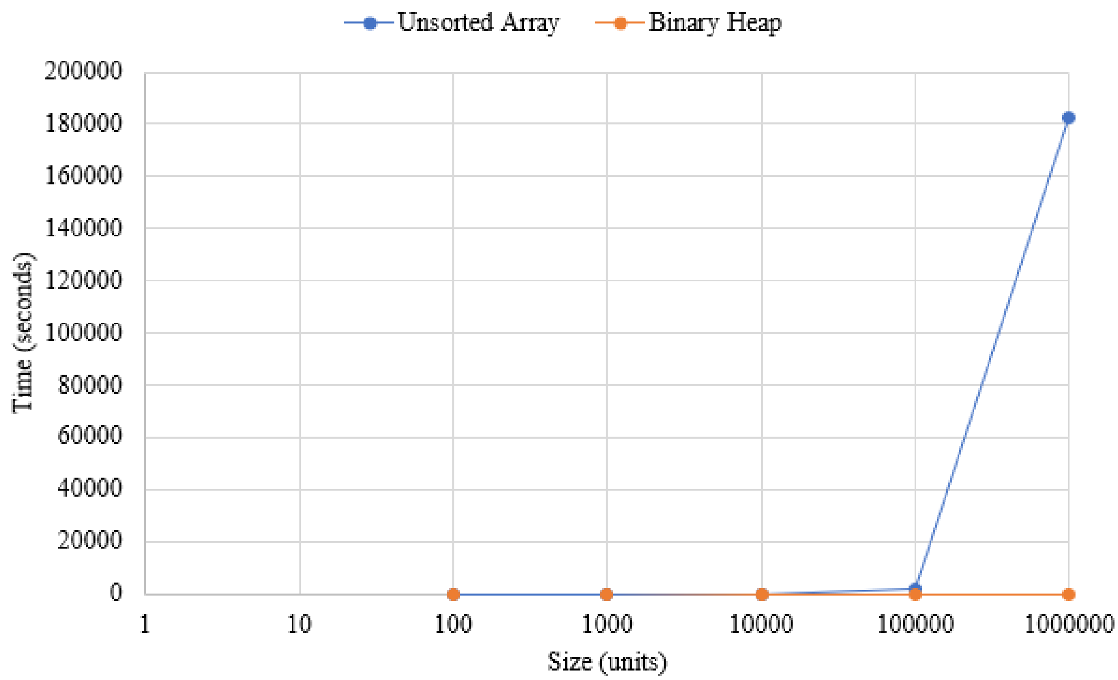

Figure 32: The Empirical Time Complexity for Unsorted vs. Binary Heap.

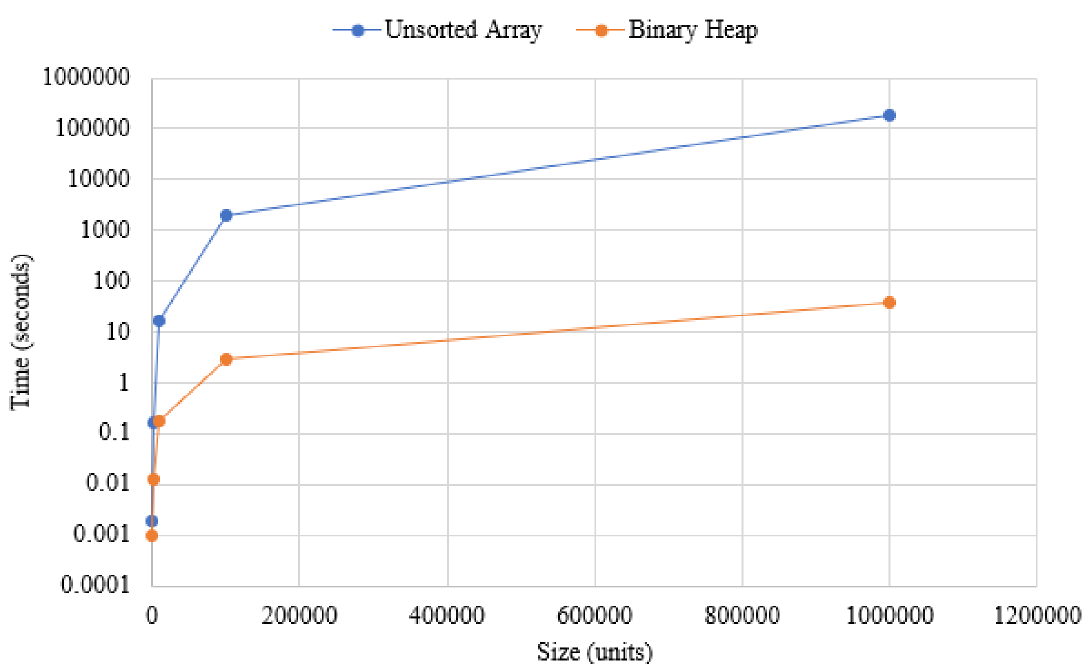

Figure 33: The Empirical Time Complexity for Unsorted vs. Binary Heap.